**Data Analysis and Visualization with Pandas in Python**

**Overview**

Students must go through and complete all instructions (in blue fonts) in this worksheet. Work must be done and saved using the Jupyter notebook (.ipynb). Submission using other file types is subjected to no grading. File naming convention is as follow:

Yourid\_Name\_Section\_Pandas.ipynb

For example,

u6519999\_Harry\_Pandas.ipynb

**Submission channel (MS Teams).**

**INTRODUCTION TO DATA VISUALIZATION WITH PYTHON**

**INSTALLING PANDAS**

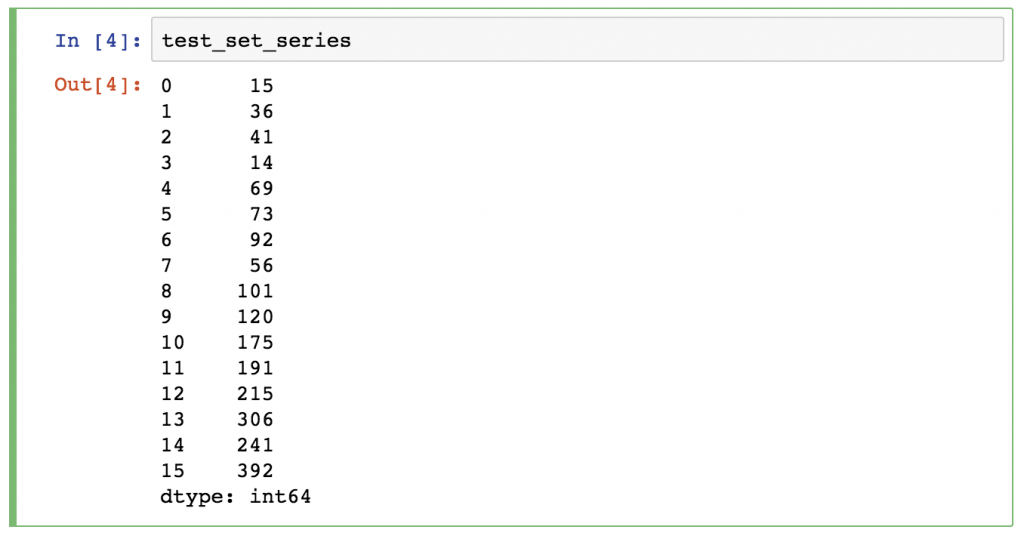
Pandas is one of the most popular Python libraries for Data Science and Analytics. In this project, you will learn the most basic and important (that is, the most often used) things that you have to know as a Data Analyst or a Data Scientist. Before you can use pandas in Jupyter notebook. You need to install Pandas libraries with the following command (either in the Jupyter notebook or Anaconda prompt).

|  |
| --- |
| pip install pandas |
|  |

After the installation is completed, the next question is how to open data files in Pandas. You might have your data in .csv files or SQL tables. Maybe Excel files. Or .csv files. Or something else. But the goal is the same in all cases. If you want to analyze that data using Pandas, the first step will be to read it into a data structure that’s compatible with Pandas. Let’s firstly understand Pandas data structures.

|  |
| --- |
| **Pandas Data Structures**  There are two types of data structures in pandas: **Series** and **Dataframes**.  **Series:** a pandas Series is a one-dimensional data structure (“a one dimensional ndarray”) that can store values — and for every value it holds a unique index, too.  **Dataframe:** a Pandas Dataframe is a two (or more) dimensional data structure – basically a table with rows and columns. The columns have names, and the rows have indexes. |

An example of Pandas series is shown below.



An example of Pandas Dataframe is shown below.



We will focus mostly on Dataframes. The reason is simple: most of the analytical methods I will talk about will make more sense in a 2D datatable than in a 1D array.

Okay, time to put things into practice! Let’s load a .csv data file into pandas!  
There is a function for it, called **read\_csv().**

Start with a simple demo data set, called zoo! This time – for the sake of practicing – you will create a .csv file for yourself! Here’s the raw data:

animal,uniq\_id,water\_need

elephant,1001,500

elephant,1002,600

elephant,1003,550

tiger,1004,300

tiger,1005,320

tiger,1006,330

tiger,1007,290

tiger,1008,310

zebra,1009,200

zebra,1010,220

zebra,1011,240

zebra,1012,230

zebra,1013,220

zebra,1014,100

zebra,1015,80

lion,1016,420

lion,1017,600

lion,1018,500

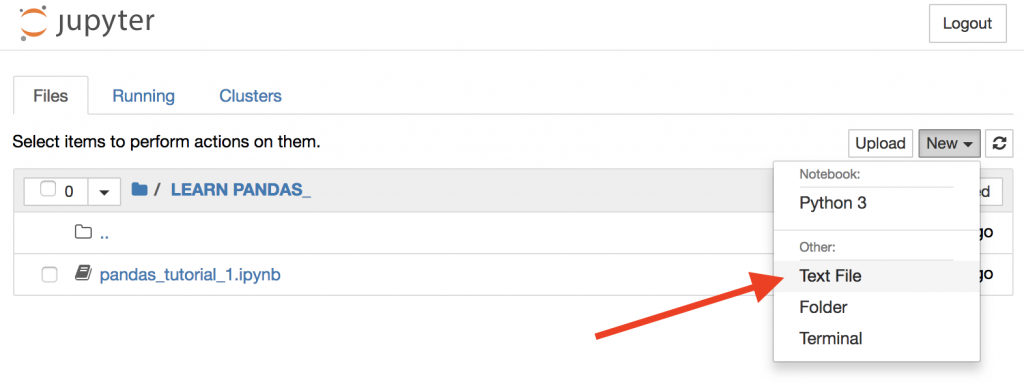
lion,1019,390

kangaroo,1020,410

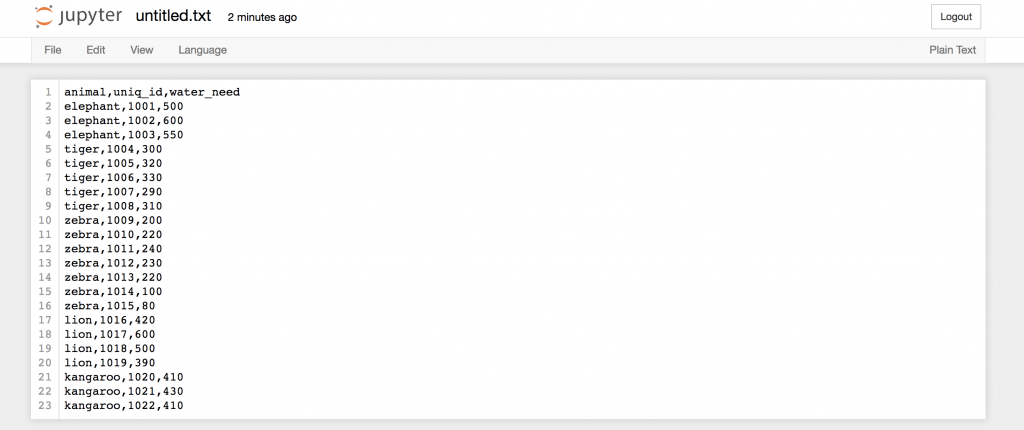
kangaroo,1021,430

kangaroo,1022,410

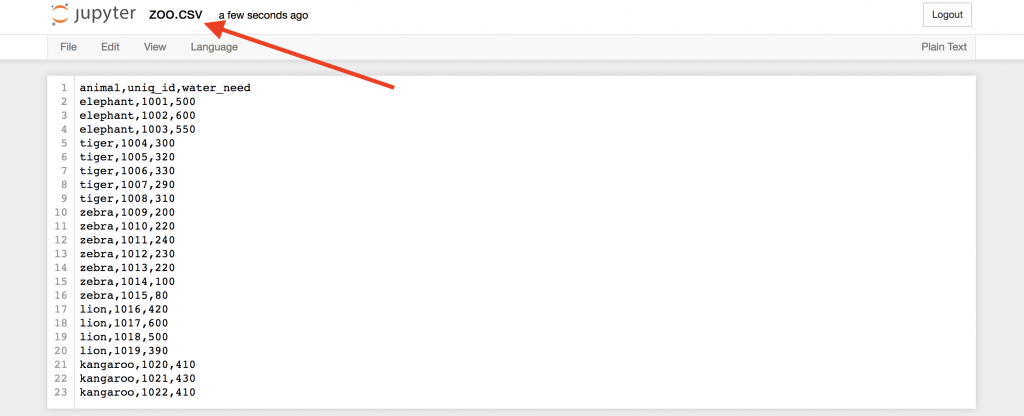
Go back to your Jupyter Home tab and create a new text file…



…then copy-paste the above zoo data into this text file…



… and then rename this text file to zoo.csv!



Okay, this is our first .csv file.

Now, go back to your Jupyter Notebook and open this freshly created .csv file in it!

Again, the function that you have to use is: read\_csv() Prior to loading .csv file, you need to import two necessary libraries as follows:



Type this to a new cell:

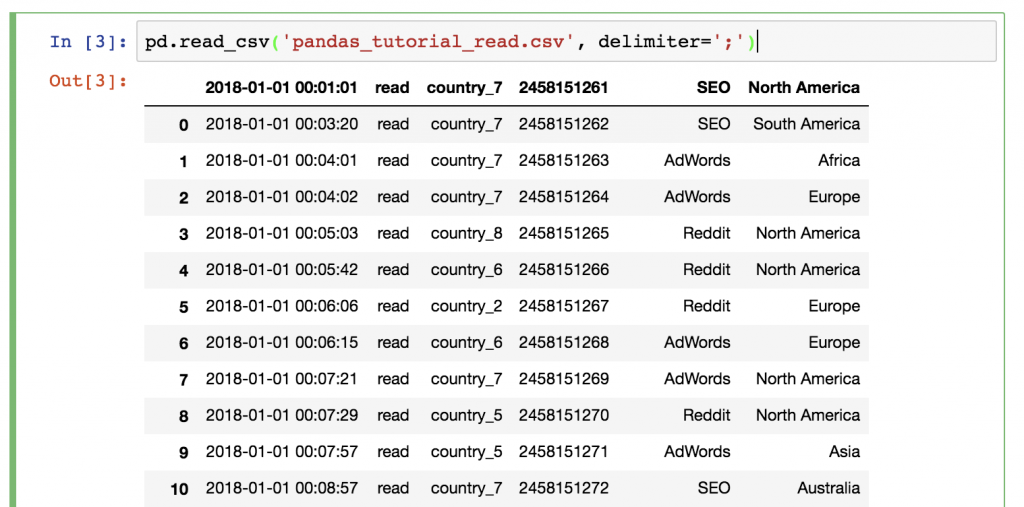
|  |
| --- |
| pd.read\_csv('zoo.csv', delimeter = ',') |
|  |



And there you go! This is the zoo.csv data file, brought to pandas. This nice 2D table? Well, this is a pandas Dataframe. The numbers on the left are the indexes. And the column names on the top are picked up from the first row of our zoo.csv file.

From the above exercises, you have learned how to create your own 2D data and saved them into .csv file. To be honest, though, you will probably never create a .csv data file for yourself, like we just did… you will use pre-existing data files.

1. **Copy pandas\_tutorial\_read.csv and save the file in the same folder that stores your .ipynb file.**
2. **Load data from pandas\_tutorial\_read.csv using pd.read\_csv() function. The following result is expected.**



Does something feel off? **Yes, this time we didn’t have a header in our .csv file**, so we have to set it up manually! (since we may not want to modify the original .csv file) Add the names parameter to your function!

The following names parameter can be added while we call pd.read\_csv() function.

|  |
| --- |
| pd.read\_csv('pandas\_tutorial\_read.csv', delimiter=';', names = ['my\_datetime', 'event', 'country', 'user\_id', 'source', 'topic'])  #This script must be on the same line. Due to a limited space on #this page, this script therefore is shown on two lines. |
|  |

The most basic method is to print your whole Dataframe to your screen. Of course, you don’t have to run the pd.read\_csv() function **again** and **again** and **again**. Just store its output the first time you run it!

[Note: pd is the loaded Pandas libraries (module) and .read\_csv() is one of the functions in the libraries.]

|  |
| --- |
| article\_read = pd.read\_csv('pandas\_tutorial\_read.csv', delimiter=';', names = ['my\_datetime', 'event', 'country', 'user\_id', 'source', 'topic']) |

After that, you can call this article\_read **(a variable name with Dataframe type)** value anytime to print your Dataframe!

1. **Use the above script to load the data from pandas\_tutorial\_read.csv file into article\_read. Then type article\_read in the next cell to observe data on the screen.**

Sometimes, it’s handy not to print the whole Dataframe and flood your screen with data. When a few rows is enough, you can print only the first 5 lines – by typing: You can specify a number of lines to be shown by adding parameter head = 10 (10 rows) in the following function.

|  |
| --- |
| article\_read.head() # by default, the first five rows (records) are shown. |

1. **Show the data for the first 5, 10 and 15 rows.**

You can show a few last rows using article\_read.tail() or show a few random rows using article\_read.sample(5)

1. **Show the last 20 rows and any 15 random rows.**

**SELECT SPECIFIC COLUMNS OF YOUR DATAFRAME**

This one is a bit tricky! Let’s say you want to print the ‘country’ and the ‘user\_id’ columns only. You should use this syntax:

|  |
| --- |
| article\_read[['country', 'user\_id']] |

**Any guesses why we have to use double bracket frames? It seems a bit over-complicated,** but maybe this will help you remember: the outer bracket frames tell pandas that you want to select columns, and the inner brackets are for the list (*remember? Python lists go between bracket frames*) of the column names.

By the way, if you change the order of the column names, the order of the returned columns will change, too:

1. **Try to show data from other columns (also try with different orders)**

*Note: Sometimes (especially in predictive analytics projects), you want to get Series objects instead of Dataframes. You can get a Series using any of these two syntaxes (and selecting only one column):*

|  |
| --- |
| article\_read.user\_id #or  article\_read['user\_id'] |

**HOW TO FILTER FOR SPECIFIC VALUES IN YOUR DATAFRAME**

If the previous one was a bit tricky, this one will be really tricky!

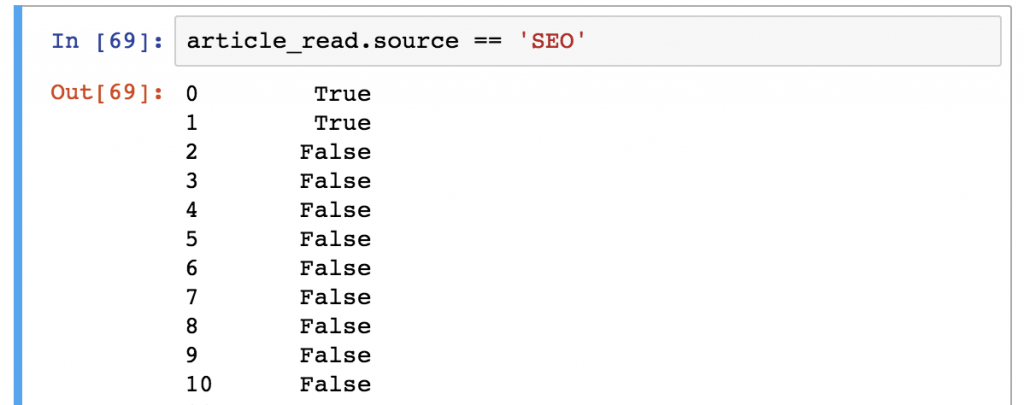
Let’s say, you want to see a list of only the users who came from the ‘SEO’ source. In this case you have to filter for the ‘SEO’ value in the ‘source’ column:

|  |
| --- |
| article\_read[article\_read.source == 'SEO'] |

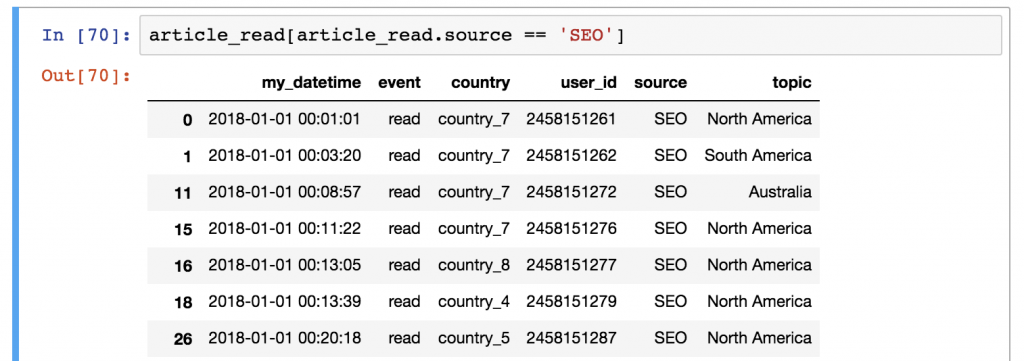
It’s worth it to understand how pandas thinks about data filtering: If we only type

article\_read.source == 'SEO'

**STEP 1)** First, between the bracket frames it evaluates every line: is the article\_read.source column’s value 'SEO' or not? The results are Boolean values (True or False).



**STEP 2)** Then from the article\_read table, it prints every row where this value is True and doesn’t print any row where it’s False as shown below.



1. **Show all records where source is from Reddit.**
2. **Show all records where article read is about Europe.**

**FUNCTIONS IN PANDAS CAN BE USED AFTER EACH OTHER**

It’s very important to understand that Pandas’s logic is very linear (So if you apply a function, you can always apply another one on it. In this case, the input of the latter function will always be the output of the previous function.

For example, you can combine these two selection functions in one script:

|  |
| --- |
| article\_read.head()[['country', 'user\_id']] |

This line first selects the first 5 rows (using .head())of our data set. And then it takes only the ‘country’ and the ‘user\_id’ columns (using [['country', 'user\_id']])

Could you get the same result with a different chain of functions? Of course you can:

|  |
| --- |
| article\_read [['country', 'user\_id']].head() |

**In this version, you select the columns first, then take the first five rows. The result is the same** – the order of the functions (and the execution) is different.

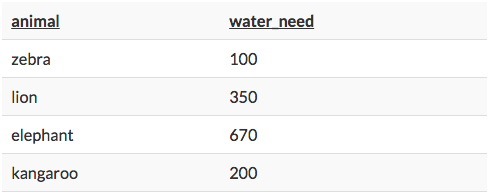
1. **Show the first five records of user\_id, country and topic for the users who are from ‘country\_2’. (Hint: you need to filter for ‘country\_2’ and select only user\_id, country, topic, and then call .head() function)**
2. **Show the first ten records of user\_id, source and topic where source is from ‘Reddit’ and topic is ‘Asia’.**

|  |
| --- |
| Solution to exercise 10.  pandas tutorial 30 - test yourself solutions |

**DATA AGGREGATION IN PANDAS**

In this section, you will be introduced to aggregation (such as min, max, sum, count, etc.) and grouping. Both are very commonly used methods in analytics and data science projects – so make sure you go through every detail in this article!

Data aggregation is the process of turning the values of a dataset (or a subset of it) into one single value. Let me make this clear! If you have a Dataframe like… (let’s get back to our zoo.csv)



…then a simple aggregation method is to calculate the summary of the water\_needs, which is 100 + 350 + 670 + 200 = 1320. Or a different aggregation method would be to count the number of the animals, which is 4. So the theory is not too complicated. Let’s see the rest in practice…

Let’s load zoo.cvs data from a file and store this Dataframe into a variable called zoo.

|  |
| --- |
| zoo = pd.read\_csv('zoo.csv', delimiter = ',') |

Okay, let’s do five things with this data:

1. Let’s count the number of rows (the number of animals) in zoo!
2. Let’s calculate the total water\_need of the animals!
3. Let’s find out which is the smallest water\_need value!
4. And then the greatest water\_need value!
5. And eventually the average water\_need!

Counting the number of the animals is as easy as applying a count function on the zoo Dataframe using zoo.count(). Actually, the .count() function counts the number of values in each column. In the case of the zoo dataset, there were 3 columns, and each of them had 22 values in it.



1. **Show the count value of animal column only.**
2. **Show the count value of animal and water\_need columns.**

Following the same logic, you can easily sum the values in the water\_need column by typing:

zoo.water\_need.sum()

1. **Try zoo.water\_need.sum()**
2. **Find the sum for animal column.**
3. **Find max and min values of water\_need.**

In Pandas, you can find Mean and Median using .mean() and .median(), respectively.

1. **Find mean and median values of water\_need.**

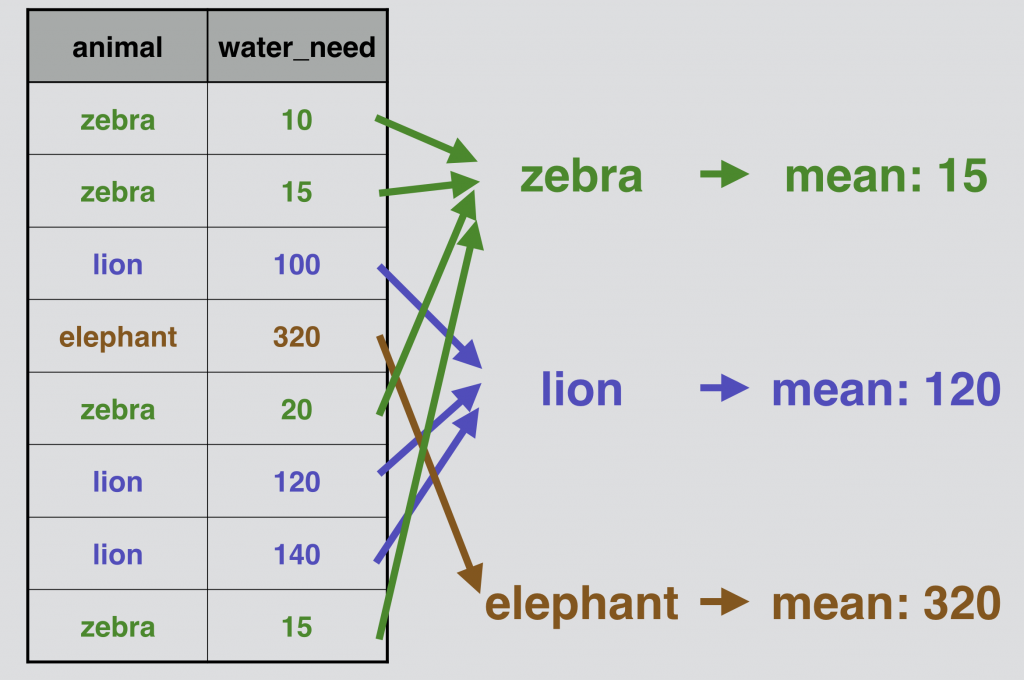
**[The correct mean and median values are 347.7272 and 325.0, respectively.]**

**DATA GROUPING IN PANDAS**

As a Data Analyst or Scientist you will probably do segmentations all the time. For instance, it’s nice to know the mean water\_need of all animals (we have just learned previously that it’s 347.72).

But very often it’s much more actionable to break this number down – let’s say – by animal types. With that, we can compare the species to each other – or we can find outliers.

Here’s a simplified visual that shows how pandas performs “segmentation” (grouping and aggregation) based on the column values!

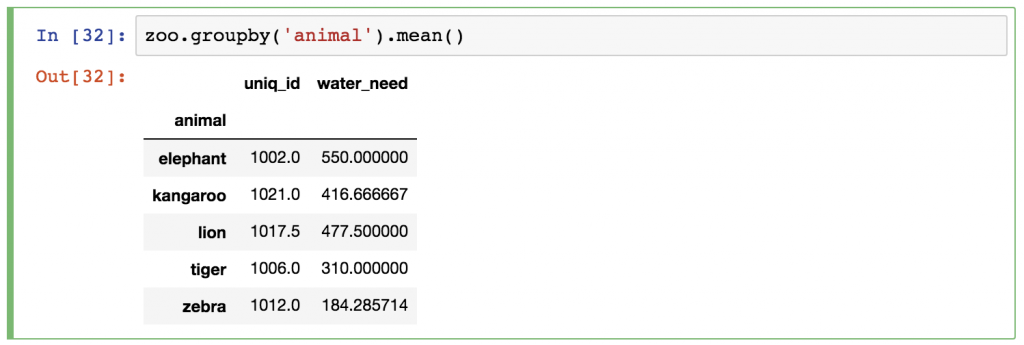


## Pandas .groupby in action

Based on last week’s exercises, let’s do the above presented grouping and aggregation for real, on our zoo Dataframe! **Note that if you start a new Jupyter notebook, you must again load a data from zoo.csv into a Dataframe variable called zoo before proceeding. If you continue from a previous notebook, make sure that you run all cells once again.**

We have to fit in a groupby keyword between our zoo variable and our .mean() function:

zoo.groupby('animal').mean()



Just as before, pandas automatically runs the .mean() calculation for all remaining columns (the animal column obviously disappeared, since that was the column we grouped by). **You can either ignore the uniq\_id column, or you can remove it afterwards by using one of these syntaxes:**

|  |
| --- |
| zoo.groupby('animal').mean()[['water\_need']] |
| This one returns a Dataframe object (value). |

If you want a series object, you must use the following syntax:

|  |
| --- |
| zoo.groupby('animal').mean().water\_need |
|  |

**Important to READ.**

Once again Pandas command is linear. **1)** zoo is your selected Dataframe. Then **2)** you group data by .groupby() function. Inside .groupby(), **3)** you specify your target column, which is ‘animal’. After that **4)** a mean value is calculated with .mean(). Finally **5)** you specify how you want to see your data (as a Dataframe or a series)

**Time to test your understanding. Load data from pandas\_tutorial\_read.csv to article\_read and complete the following exercises:**

1. **Find the most frequent source in the article\_read Dataframe. (Hint: you need .groupby() for ‘source’ column and count them. The correct answer is Reddit!!!!)**
2. **For the users coming from ‘country\_2’, what is the most frequent topic and source combined? [Hint: Step 1: you need to filter for only ‘country\_2’. Step 2: you need to group ‘topic’ and ‘source’. Step 3: apply .count( )]**

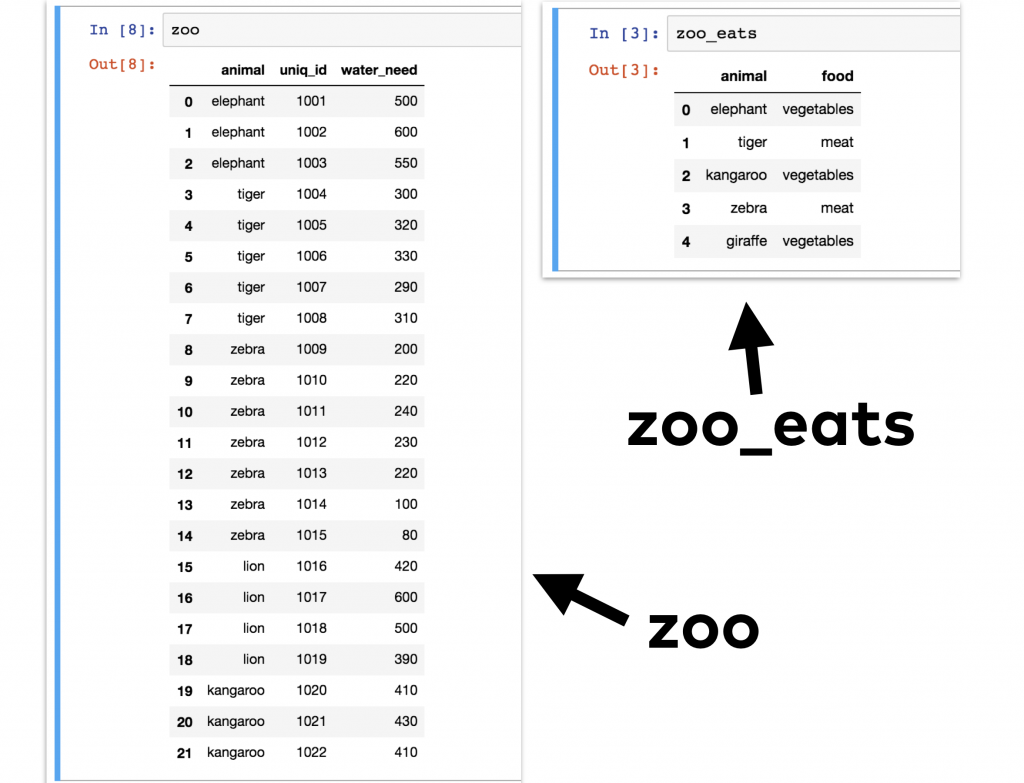
**Correct answer is the most frequently read topic is ASIA and the source is from Reddit (139).**

**DATA MERGING IN PANDAS**

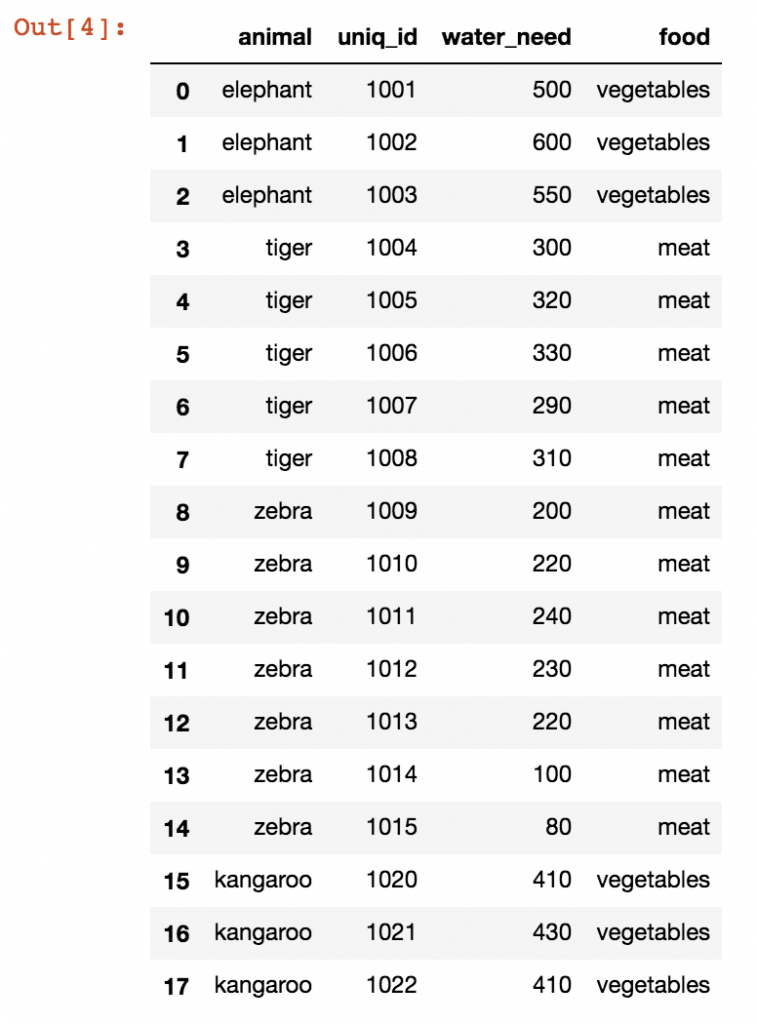
In real life data projects, we usually don’t store all the data in one big data table. We store it in a few smaller ones instead. There are many reasons behind this; by using multiple data tables, it’s easier to manage your data, it’s easier to avoid redundancy, you can save some disk space, you can query the smaller tables faster, etc.

The point is that it’s quite usual that during your analysis you have to pull your data from two or more different tables. The solution for that is called **merge**.

Let’s take our zoo Dataframe in which we have all our animals… and let’s say that we have another Dataframe, zoo\_eats, that contains information about the food requirements for each species. [Note: both are available in Teams]



We want to merge these two pandas Dataframes into one big Dataframe. Something like this:



This can easily be done by using .merge() as shown below.

|  |
| --- |
| zoo.merge(zoo\_eats) |
|  |

**[Note: Originally, there are 22 records in zoo. However, after merging only 18 records remain. Can you guess what is happening here?]**

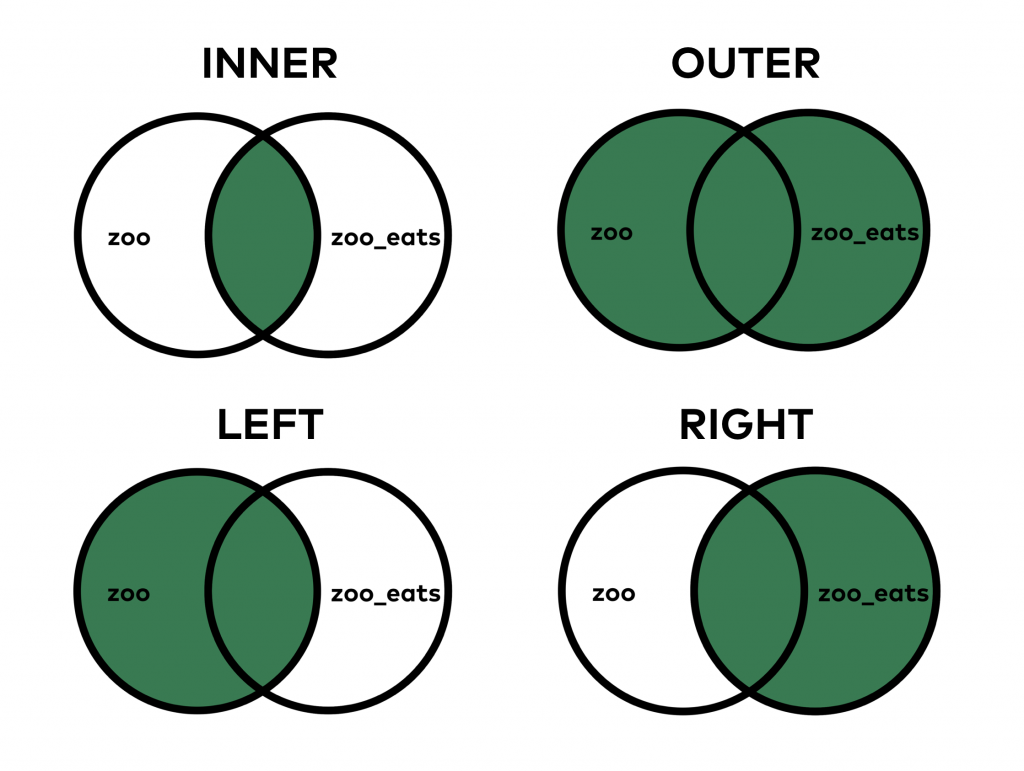
First, I specified the first Dataframe (zoo), then I applied the .merge() pandas method on it and as a parameter I specified the second Dataframe (zoo\_eats). I could have done this the other way around:

|  |
| --- |
| zoo.merge(zoo\_eats)  is symmetric to  zoo\_eats.merge(zoo) |
|  |

1. **Try zoo\_eats.merge(zoo) and zoo.merge(zoo\_eats)** **and observe the results. You must load a data from zoo\_eats.csv before trying the following code.**

As you can see, the basic merge method is pretty simple. Sometimes you have to add a few extra parameters though.

One of the most important questions is ***how*** you want to merge these tables. In SQL, we learned that there are different JOIN types.



When you do an INNER JOIN (that’s the default both in SQL and pandas), you merge only those values that are found in **both tables**. On the other hand, when you do the OUTER JOIN, it merges all values, even if you can find some of them in only one of the tables.

To specify how we are going to merge data, a syntax ‘how = ’ is needed.

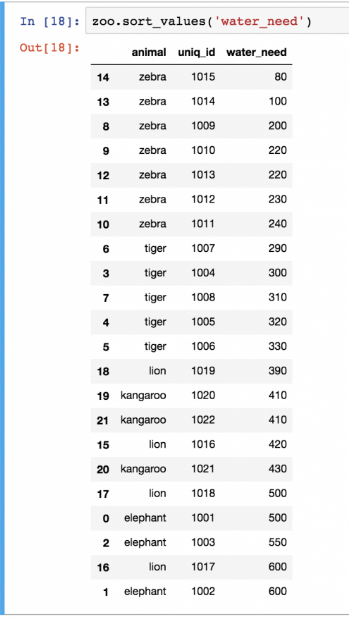
Try the following scirpts and observe the results.

1. **zoo.merge(zoo\_eats, how = 'outer')**
2. **zoo.merge(zoo\_eats, how = 'left')**
3. **zoo.merge(zoo\_eats, how = 'right')**

**DATA SORTING IN PANDAS**

Sorting is essential. The basic sorting method is not too difficult in pandas. The function is called sort\_values() and it works like this:

|  |
| --- |
| zoo.sort\_values('water\_need') |
|  |



The only parameter I used here was the name of the column I want to sort by, in this case the water\_need column. Quite often, you have to sort by multiple columns, so in general, I recommend using the by keyword for the columns:

|  |
| --- |
| zoo.sort\_values(by = ['animal', 'water\_need']) |
|  |

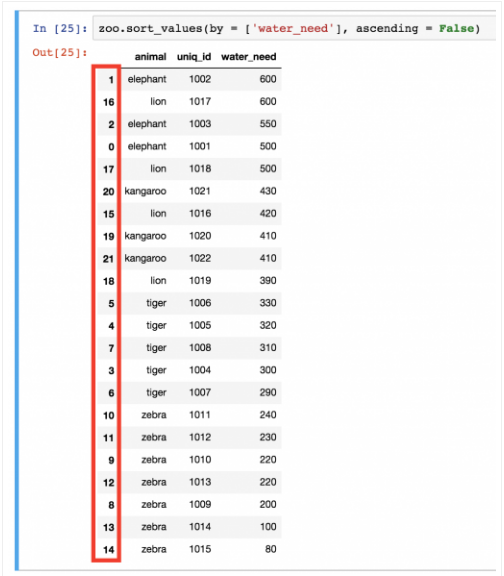
1. **Try the above Python codes.**
2. **Swap positions of ‘animal’ and ‘water\_need’, and observe the result.**
3. **Show only animal and water\_need columns. (water\_need column first)**

Note: you can use the by keyword with one column only, too, like zoo.sort\_values(by = ['water\_need']).

sort\_values sorts in ascending order, but obviously, you can change this and do descending order as well:

|  |
| --- |
| zoo.sort\_values(by = ['water\_need'], ascending = False) |
|  |

**Try the above Python code. You should get the following output:**



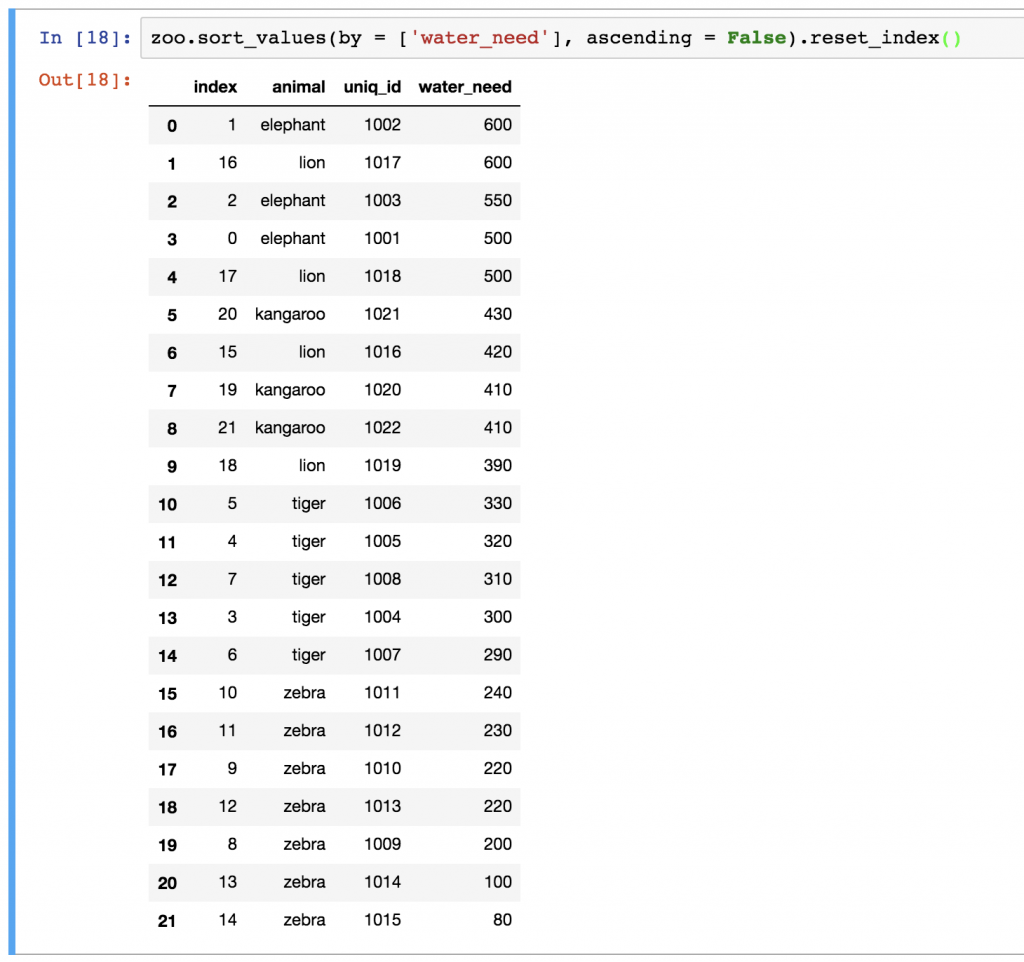
**What a mess with all the indexes after that last sorting, right?**

It’s not just that it’s ugly… wrong indexing can mess up your visualizations or even your machine learning models.

The point is: in certain cases, when you have done a transformation on your Dataframe, you have to re-index the rows. For that, you can use the reset\_index() method. For instance:

|  |
| --- |
| zoo.sort\_values(by = ['water\_need'], ascending = False).reset\_index() |
|  |

1. **Try the above Python code. You should get.**



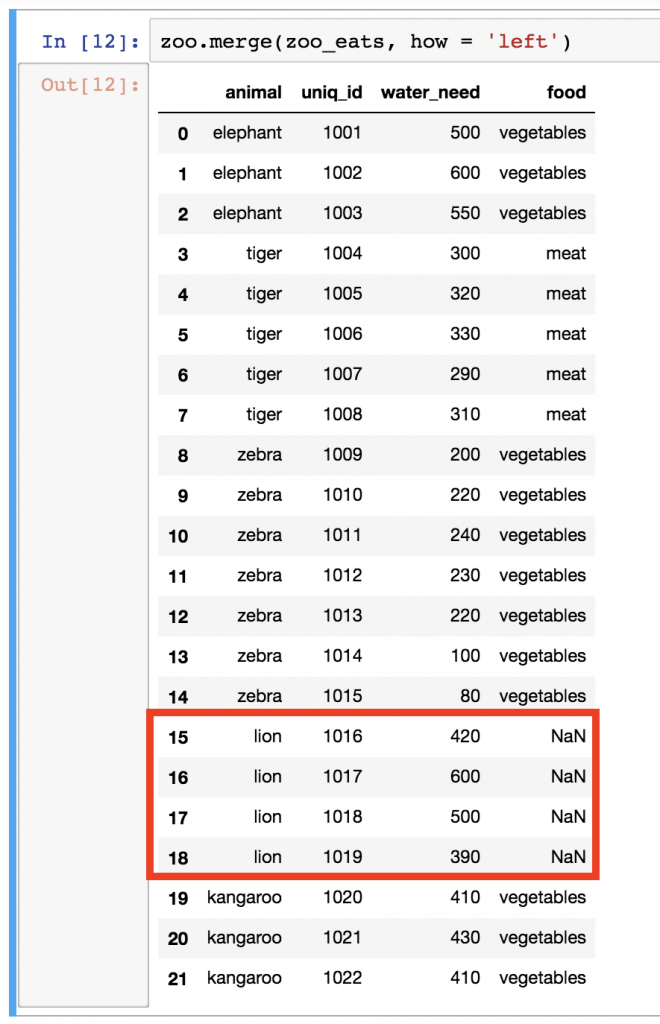
As you can see, our new Dataframe kept the old indexes, too. If you want to remove them, just add the drop = True parameter in .reset\_index():

1. **Add the above parameter in .reset\_index() and observe the result.**

DATA MUNGING (CLEANSING) in PANDAS

Let’s rerun the left-merge method that we have used above:

zoo.merge(zoo\_eats, how = 'left')



Remember? These are all our animals. The problem is that we have NaN values for lions. NaN itself can be really distracting, so I usually like to replace it with something more meaningful. In some cases, this can be a 0 value, or in other cases a specific string value, but this time, I’ll go with unknown. Let’s use the fillna() function, which basically finds and replaces all NaN values in our Dataframe:

1. **Try the below Python code and see the results.**

zoo.merge(zoo\_eats, how = 'left')**.fillna('unknown')**

Note: since we know that lions eat meat, we could have filled?

1. **Fill in the appropriate food for lion (meat or vegetables????). Change from ‘Unknown’ to an appropriate food.**

**INTRODUCTION TO DATA VISUALIZATION in PANDAS**

Start by loading these modules to your jupyter notebook.

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

Here we are going to create our own Dataframe by specifying the following data values.

df **=** pd**.**DataFrame({

'name':['john','mary','peter','jeff','bill','lisa','jose'],

'age':[23,78,22,19,45,33,20],

'gender':['M','F','M','M','M','F','M'],

'state':['california','dc','california','dc','california','texas','texas'],

'num\_children':[2,0,0,3,2,1,4],

'num\_pets':[5,1,0,5,2,2,3]

})

Try the above Python code and the following result should be obtained.

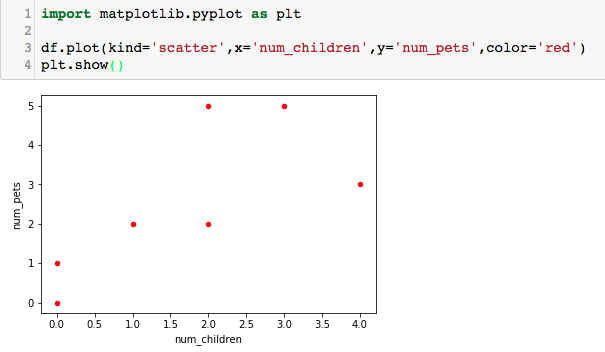


Pandas has **tight integration** with matplotlib.

You can plot data directly from your Dataframe using the [plot()](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html) method:

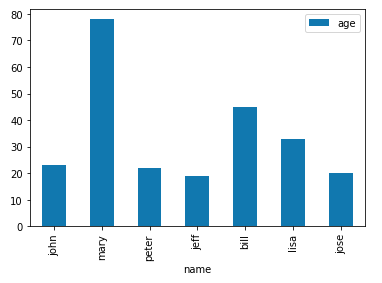
1. **Try the following .plot()**  Note: If you have already imported matplotlib in any of the previous cells, you do not need to import it again.

Now you will see a graph showing a number of children in x-axis, where y-axis is a number of children (for each record).

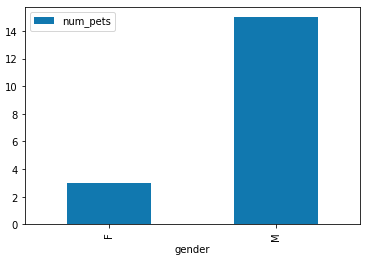


x 🡪 x-axis and y 🡪 y-axis

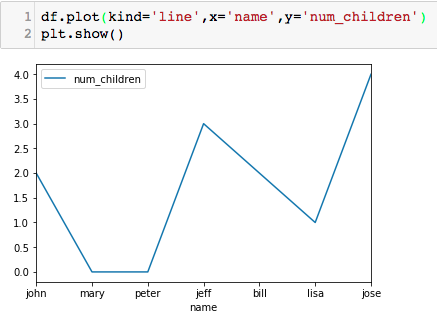
1. **Try the above .plot() with x = ‘age’**
2. **Try .plot() with kind = ‘bar’ and change x-axis to ‘name’ and y-axis to ‘age’ (expected Bar graph is as follows)**

****

1. **With the above .plot(), change kind to ‘barh’ and observe the printed Bar graph.**
2. **Try .plot() with different values for x-axis and y-axis. For example, how many children does each person have? How many pets does each person own?**
3. **Plot the following graph (No. of pets classified by a gender). Hint: you need to use .groupby() based on gender then plot the graph.**



Next you will learn how to do Line plot with multiple columns. Try the following code.



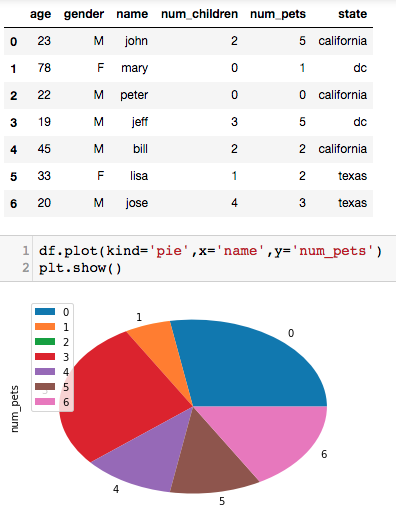
*plot()*takes an optional argument 'ax' which allows you to reuse an Axis to plot multiple lines. We need to put ax = plt.gca(), where gca stands for get current axis. .gcs() tells matplotlib to reuse the existing axis to plot other lines. Try the following code.



We can save graphs by calling plt.savefig('outputfile.png'), instead of calling plt.show() to show the graph.

1. **Save one of the previous graph in your working directory and check your directory for the saved file. Check your folder if the saved file exists.**

You can also plot a Pie chart by changing kind to ‘pie’. With the following Dataframe, try the given Python code.



As you may observe from the graph, its Legend is index values. It might cause confusion for person to understand it. It becomes necessary to define a label, which can be done as follows: (We no longer need x-axis in .plot() as it will be replaced with labels.)



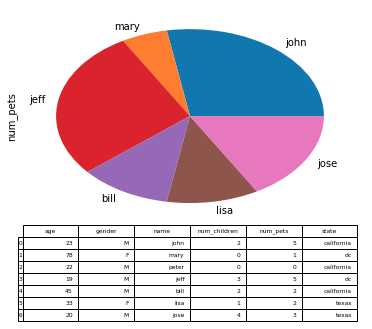
1. **Try the above code.**

The legend shown in the square box may be displayed on top of Pie chart, if you want to remove that it can be done by adding legend = False in the .plot()

1. **Put “legend = False” as one of the argument in .plot() and observe the result.**

You can add Dataframe as a table together with a graph by adding table = df (df is your Dataframe variable; if your Dataframe variable is zoo, then it must be table = zoo).

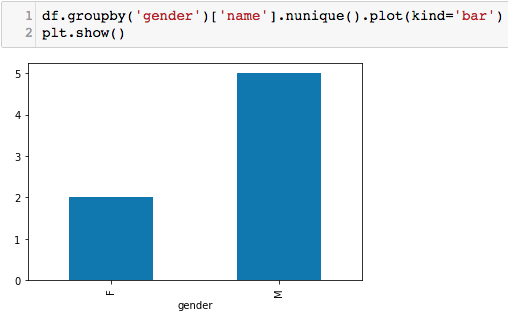
1. **Put “table = df” in .plot() (Expected output is as follows.)**

****

If you want to adjust a size, it can be done using figsize = (10,10) [Note that values (10,10) in the blanket are the width and height values]

1. **Make a graph bigger and smaller.**

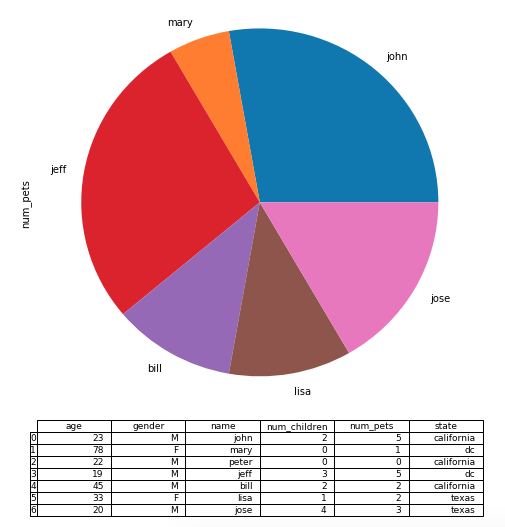
We can plot a graph based on a group of data using .groupby(). For example,



The above Bar graph shows data based on groups of gender, and ‘name’ is defined as a key index which is a unique value [ .nunique() ]. Note that if values in ‘name’ column are not unique, another column must be used.

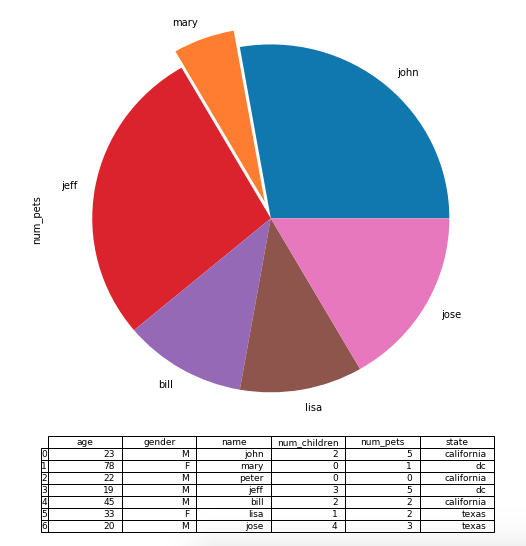
1. **With reference to the above example, change to Pie graph.**
2. **Group data by state and plot Pie and Bar graphs**

From exercise 41, you might observe different in image viewing angle. That does not affect the result.

****

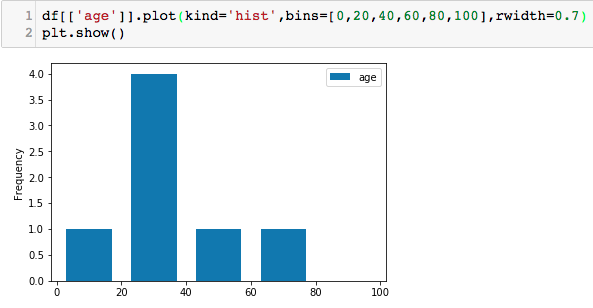
We can highlight certain pies by adding explode = (0,0.1,0,0,0,0,0) in .plot(). The order is anti-clockwise and the first pie starts at 180 degree. In this case, the first pie is John and we want to explode the second pie, which is ‘Mary’.

You might wonder why there are 7 values for an argument ‘explode’. This is because there are 7 people, however ‘Peter’ does not have a pet so, a pie piece for ‘Peter’ is there with a size of 0. It is invisible.



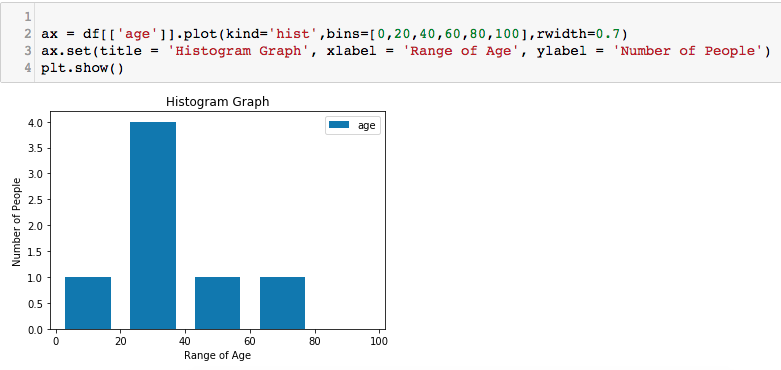
1. **Highlight Jeff and Jose pies.**

The following example illustrates how to plot Histogram based on ‘age’. By default, x-axis and y-axis do not show any labels.

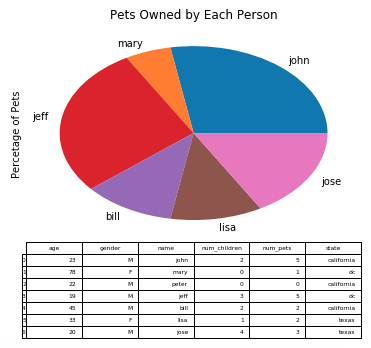


We can define a label for each axis and a title of the graph by using .set() as shown below.

Since we are going to define a label for x- and y-axis, we must execute a code and store a value in one variable. In this example it is ‘ax’.



1. **From the Pie graph in exercises 41, set a title of the graph to ‘Pets Owned by Each Person’ and set a label in y-axis to ‘Percentage of Pets’. Expected result**

****

**Classwork**

1. **Based on a given data sets (car\_final.csv and car\_insurance.csv), complete the following tasks:** 
   1. **Show the first 5 records of each data set.**
   2. **Merge car\_final.csv and car\_insurance.csv. Note that car\_final.csv is the main data set.**
   3. **Sort the data by car (alphabetically)**
   4. **Plot a bar graph showing, for each car type, number of cars that has moderate insurance fee.**
   5. **Plot a histogram for ‘maintenance’ and define bin = 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000**
2. **Based on a given data sets (zoo\_final.csv and zoo\_weight.csv), complete the following tasks:** 
   1. **Show the first 5 records of each data set.**
   2. **Merge zoo\_final.csv and zoo\_weight.csv. Note that zoo\_final.csv is the main data set.**
   3. **Replace NaN values with appropriate values.**
   4. **Plot a bar graph showing, for each animal type, a number of animal that has a moderate weight**
   5. **Change title of a graph to “Animal by Weight” and labels of y-axis to “Amount” and x-axis to “Animal Type”.**

**(OPTIONAL) PLOTTING DATA ON A MAP**

Visualizing data on a map sometimes is very important and useful for a strategic decision making.

The following exercises deal with Scottish’s hill data (scottish\_hills.csv). Main data in this .csv file contains latitude, longitude and height of scottish’s hills.

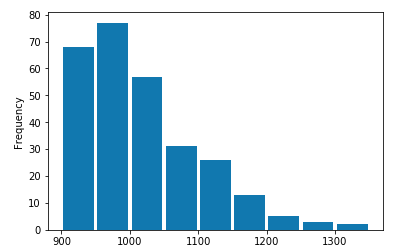
****

1. **Load the data from scottish\_hills.csv to a Dataframe variable called *sh*. For example,**

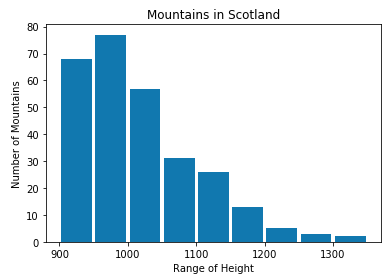
**sh = pd.read\_csv…..**

1. **Plot a Histogram for ‘height’ and define bin = 900,950,1000,1050,1100,1150,1200,1250,1300,1350, with rwidth = 0.9**

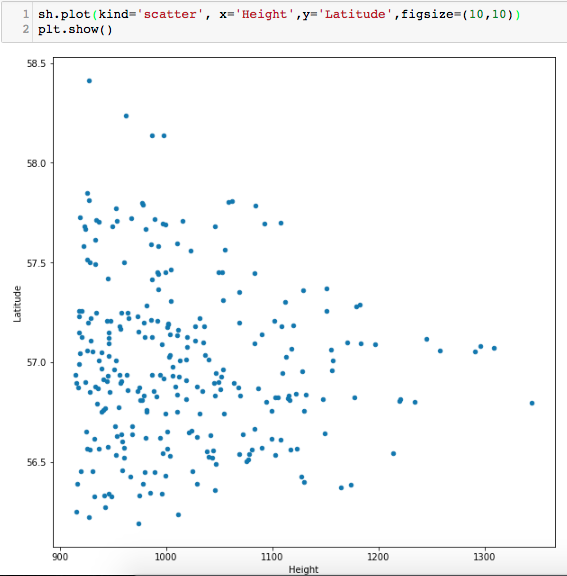
**The expected graph is shown below.**

****

1. **Set a title and labels for each axis as follows.**

****

Next, we are going to do a scatter plot to show the relationship between ‘height’ and ‘latitude’. Try the following code to see the graph.



1. **Set a title to ‘Mountains in Scotland’ and label for x-axis to “Mountains’ Height”.**

We can set a filter to show plots for specific values. For example, if you want to see only hills with a height of 1,200 meters or higher. We can use the following code.

ax = sh[sh.Height > 1200].plot(kind='scatter', x='Height',y='Latitude',figsize=(10,10))

1. **Plot Scatter graph with hills’ height lower than 1,100 meters. Make sure that graph’s title and label are set as instructed in exercise 4.**
2. **Plot Scatter graph with latitude lower than 57 degrees.**

We can also specify a range of data to be plotted. For example, if we want to plot only hills with a height between 900 and 1000 meters. We need to do the following things. First, we specify a height (lower bound) and store our results in another datagrame, such as sh1. (We do not plot here)

sh1 = sh[sh.Height > 900] # here we filter data once before plotting

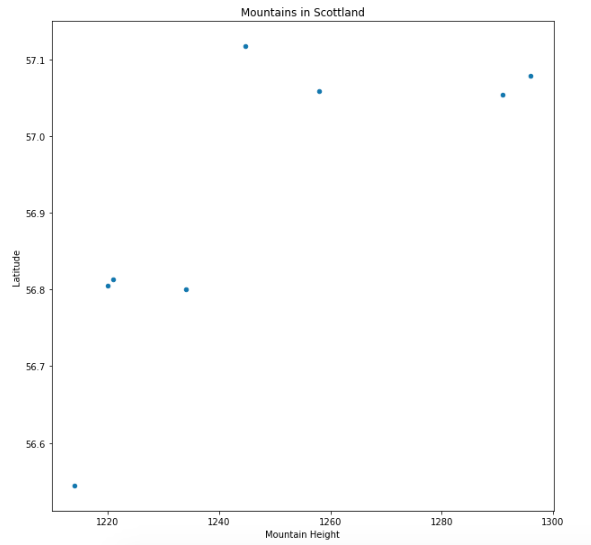
Then we do it again with an upperbound value and store the results in another Dataframe such as sh2. (Now we plot the data)

sh2 = sh1[sh1.Height < 1100].plot(………..)

sh2.set(title = ………)

Then run plt.show()

1. **Do Scatter plot with a height between 1,200 and 1,300 meters. The expected graph is shown below.**

****

1. **Scatter plot with latitude between 57 and 58 degrees.**

Since we do have both latitude and longitude values in scottish\_hills.csv, we can show the exact location of hills on Scotland map. To do so, we need to import cartopy (another Pandas’s Library).

First thing first, since your environment may not have this package, you need to install the package prior to using it. Go to Anaconda’s command prompt (if it does not work, you may need to run Anaconda’s command prompt with Admin’s right) and

conda install -c conda-forge cartopy

Then use the given python code below to plot locations of hills on the map.

import cartopy.crs **as** ccrs

from cartopy.mpl.ticker import LongitudeFormatter, LatitudeFormatter

import cartopy.feature **as** cfeature

plt**.**figure(figsize**=**(20,10))

ax **=** plt**.**axes(projection**=**ccrs**.**Mercator())

ax**.**coastlines('10m')

ax**.**xaxis**.**set\_visible(True)

ax**.**yaxis**.**set\_visible(True)

ax**.**set\_yticks([56,57,58,59], crs**=**ccrs**.**PlateCarree()) #Ranges of Latitude

ax**.**set\_xticks([**-**8, **-**6, **-**4, **-**2], crs**=**ccrs**.**PlateCarree()) #Ranges of Longitude

lon\_formatter **=** LongitudeFormatter(zero\_direction\_label**=**True)

lat\_formatter **=** LatitudeFormatter()

ax**.**xaxis**.**set\_major\_formatter(lon\_formatter)

ax**.**yaxis**.**set\_major\_formatter(lat\_formatter)

ax**.**set\_extent([**-**8, **-**1.5, 55.3, 59]) #boundaries of graph based on Long / Lat

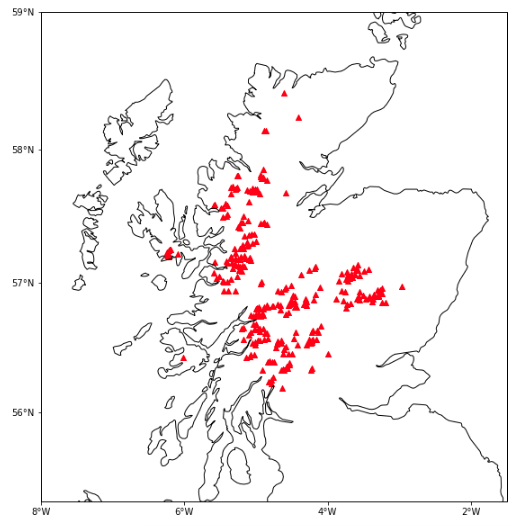
plt**.**scatter(sh['Longitude'],sh['Latitude'],

color**=**'red', marker**=**'^', transform**=**ccrs**.**PlateCarree())

plt.show()

#plt**.**savefig("hillsdata.png")

With the above codes, the expected graph is as follows.



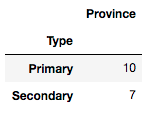
**From previous exercises, you have learned how to do a scatter plot on a map (using cartopy package). Let’s try to plot something on Thailand map.**

**Load the data from Province.csv to pr Dataframe and from ProvinceTax.csv to prt Dataframe (Note that in this sample data set, it contains data for 17 provinces only.**

1. **Show the first 10 records of primary cities, and show only ‘Province’ and ‘Latitude’ columns. The expected output is**



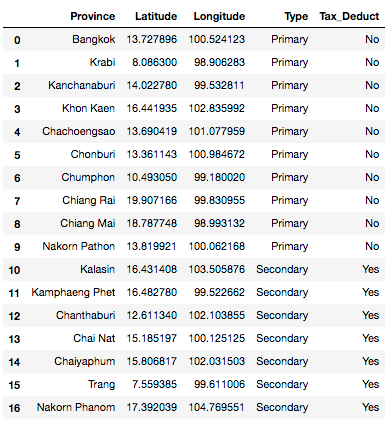
1. **Show a total number of primary and secondary cities (Province column only). The expected output is**

****

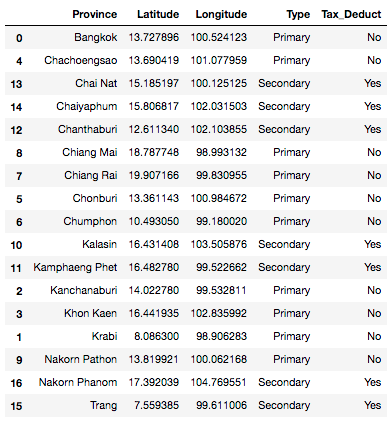
1. **Merge prt to pr and store the results in a Dataframe called *pr\_merge*.**

***pr\_merge = pr.merge…..***

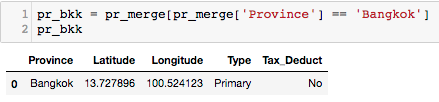
**The expected output is**

****

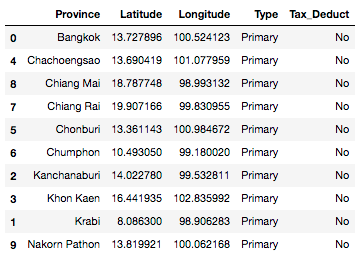
1. **Sort the values by Province and store the results in a Dataframe called *pr\_merge\_sort*, the expected output is as follows.**

****

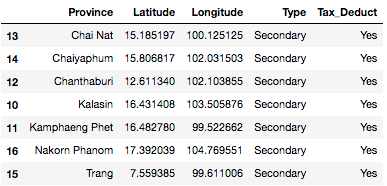
In a case that we want to select one particular record, for example Bangkok, we can use the following Python code.



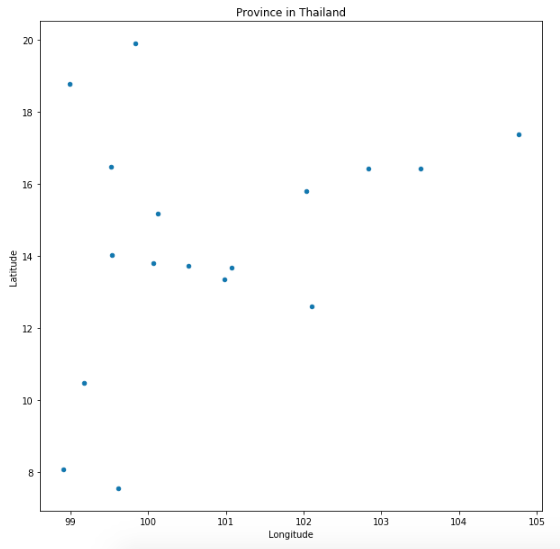
1. **Select all records where “Type” is primary and store the results in a Dataframe called *pr\_primary*. The expected output is**

****

1. **Select all records where “Type” is secondary and store the results in a Dataframe called *pr\_secondary.* The expected output is**

****

1. **Show Scatter plot with x-axis is Longitude and y-axis is Latitude. Also set a title of the graph to Province in Thailand. The expected graph is**

****

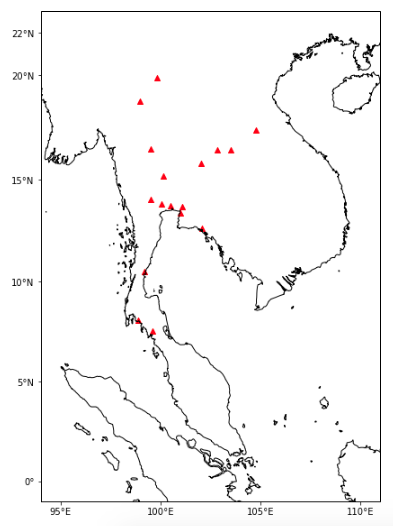
1. **Show Scatter plot on Thailand map. The expected graph is**

**(You need to adjust Longitude and Latitude ranges in the given cartopy’s code to fit Thailand’s map)**

**ax.set\_yticks**

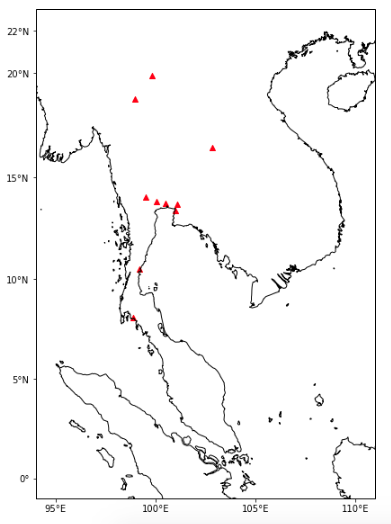
**ax\_set\_xticks**

**ax.set\_extent**

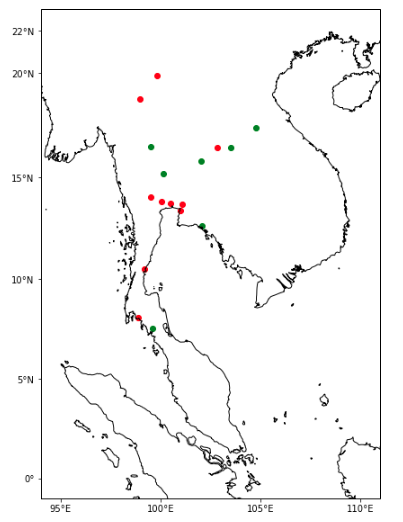
****

Based on the two Dataframes, namely pr\_primary and pr\_secondary,

1. **Show Scatter plot of primary cities only.**

****

1. **Show Scatter plot of both primary and secondary cites, where a marker is set to 'o', and a color of marker for primary and secondary cities is set to red and green, respectively. The expected graph is**

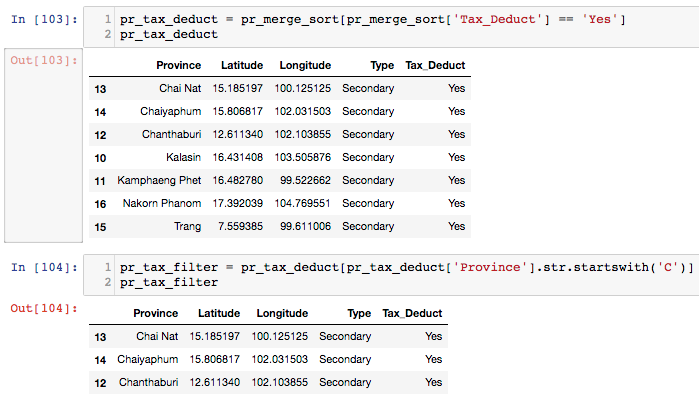
****

1. **Show a Scatter plot of Bangkok only. Note that you have previously filter this data and store it in a Dataframe variable, namely pr\_bkk.**

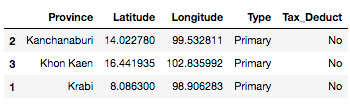
From the Dataframe, namely *pr\_merge\_*sort, that you have previously merged and sorted, we can show only cities that expenses (money spent in that city) can be tax deductible. We can also show only cities that a first character is ‘C’.

First step (cell 103 below), we should filter out the data by selecting only cities that are tax deductible. Store the filtered data in a new Dataframe variable.

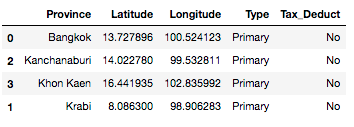
Second step (cell 104 below), from the new Dataframe variable, we use a string method called .startswith() to select cities that the first character of cities’ name is started with a character we want.



1. **Show primary cities where expenses are not tax deductible, and a city name is started with ‘K’. The expected output is**

****

1. **Show primary cities where expenses are not tax deductible, and a city name is started with ‘B’ and ‘K’. The expected output is**

****

1. **Plot the filtered data obtained in exercise 20 and 21 on a map.**